VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

POKROČILÝ EDITOR VHDL SOUBORŮ

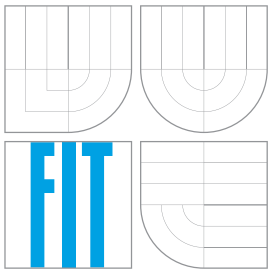
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VOJTĚCH KLIMENT

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

POKROČILÝ EDITOR VHDL SOUBORŮ

ADVANCED EDITOR OF VHDL FILES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — VOJTĚCH KLIMENT
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ NOVOTŇÁK
SUPERVISOR

BRNO 2012

Abstrakt

Tato bakálářská práce se zabývá vývojem aplikace, která umožňuje návrháři číslicových obvodů snadněji vkládat a propojovat jednotlivé komponenty VHDL entity. Práce vysvětluje základní principy jazyka VHDL a způsob, jakým se v jazyce VHDL číslicové systémy popisují. Dále jde zde nastíněna problematika syntetizovatelných šablon a jsou vysvětleny důvody jejich používání. V dalších částech práce je popsán návrh a způsob implementace editoru, který poskytuje uživateli možnost snadnější práce na návrhu pomocí grafického prostředí, které editor obsahuje.

Abstract

This bachelor's thesis deals with an evolution of an application, with which the designer of digital systems can insert and connect components of VHDL entity easier. You can find an introduction to the VHDL language here and the ways how to design digital systems with VHDL. The reasons of using synthesizable templates will be described. Next part is a description of a design and an implementation of an editor, which makes it easy to design by an graphic interface.

Klíčová slova

VHDL, syntetizovatelné šablony, editor

Keywords

VHDL, synthesizable templates, editor

Citace

Vojtěch Kliment: Pokročilý editor VHDL souborů, bakalářská práce, Brno, FIT VUT v Brně, 2012

Pokročilý editor VHDL souborů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Novotňáka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vojtěch Kliment
16. května 2012

Poděkování

Tímto bych rád poděkoval svému vedoucímu panu Ing. Jiřímu Novotňákovi za to, že mi poskytoval odbornou pomoc při řešení mojí bakalářské práce.

Obsah

Kapitola 1

Úvod

Tvorba návrhu číslicových obvodů spočívá ve vytváření komponent, popisu propojení mezi nimi nebo popisu chování jednotlivých komponent. Při tvorbě tohoto návrhu v jazyce VHDL se pak používají ověřené, ale často zdlouhavé konstrukce, jejichž vymyšlení není náročné na přemýšlení, ale jejich psaní je nutná mechanická práce zabírající čas. Cílem je tak vytvořit nástroj, který bude návrháři usnadňovat právě tuto část práce, bude mu umožňovat jednodušeji vytvářet komponenty, popisovat je nebo projovat mezi sebou.

Dále se musí návrhář číslicových obvodů oproti ostatním programátorům potýkat s jedním problémem a to je, že jeho kód bude pravděpodobně realizován hardwarem, tedy nějakým fyzickým modelem, který má na rozdíl od virtuálního modelu určitá omezení. Jeden z problémů je viděn v procesu syntézy zdrojového kódu na schéma zapojení. Tento automatický proces totiž nemusí programátora správně pochopit a může vytvořit systém, který se chová mírně odlišně[4][7]. Dalším cílem je umožnit návrháři vkládat takové konstrukce, které jsou tzv. syntetizovatelné, tzn. že budou moci být realizovány hardwarem. Editor by měl umožňovat vkládat předem připravené šablony, různě je modifikovat nebo i vytvářet svoje vlastní šablony.

Měl by tedy vzniknout program, který bude práci na tvorbě modelu usnadňovat tím, že bude za uživatele doplňovat právě ty konstrukce a úseky zdrojového kódu, které by zcela určitě dokázal rychle vymyslet, ale jejich psaní by mu zabíralo zbytečně mnoho času. Rozhodl jsem se, že bude program pracovat jako klikátko, uživatel tak bude moci vytvářet návrh systému v grafickém prostředí. Podle mého názoru je tento způsob velmi rychlý a zároveň i dostatečně intuitivní, aby byl použitelný pro většinu uživatelů.

V práci je v 2. kapitole definován cíl práce a popsán současný stav. V kapitole č. 3 je probrán jazyk VHDL, jsou zde probrány jednotlivé návrhové jednotky a způsoby, jakým lze systém popisovat. Kapitola č. 4 se zabývá charakteristikou nejčastěji používaných komponent a dále pak jejich popisem pomocí syntetizovatelných šablon. Návrh programu usnadňující práci s popisem systému je možné najít v kapitole č. 5. Implementace programu je pak popsána a vysvětlena v kapitole č. 6 a způsob testování se nachází v kapitole č. 7.

Kapitola 2

Formulace cíle a analýza současného stavu

2.1 Cílový produkt

Jak už bylo řečeno v úvodu, hlavní funkcí, kterou by měl tento projekt splňovat je to, aby ulehčoval práci při psaní těch pořád stejných řádků kódu. Je to práce, která je nenáročná, mechanická, nicméně bez ní se návrhář neobejde. Program bude právě toto dělat za něj. Aplikace uživateli nabídne možnost definovat si vlastní komponenty, které si poté bude moci ukládat a později znovu využívat.

Samozřejmostí je možnost vygenerovat si základní strukturu VHDL po vytvoření nového souboru pomocí rychle pochopitelných dialogových oken.

Dále jsem se rozhodl, že program bude obsahovat grafický panel, kam si může návrhář kreslit schéma. Schéma nebude jenom obrázek, ale právě na jeho základě bude program generovat kód. Tím se splní hned dva úkoly návrháře najednou, za prvé nebude muset kreslit návrh na papír a za druhé ani nebude muset návrh přepisovat ručně, ale využije implementovaný generátor, který za něj alespoň některé části přepíše.

Aplikace bude umožňovat uživateli vkládat syntetizovatelné šablony, které budou sice již předpřipravené, ale ani tak si je nebude muset uživatel upravovat klasicky, tedy tím, že by musel postupně všechny identifikátory přepisovat ručně. Bude mít k dispozi mnohem rychlejší systém.

Zcela určitě je nutné implementovat jednu ze základních funkcí každého editoru a to je zvýrazňovač syntaxe daného jazyka. Dále bude dbáno na to, aby zdrojový kód vypadal hezky a byl přehledný, program tedy bude vždy generovaný správně odsazovat a zarovnávat.

2.2 Současný stav

Při koupi jakéhokoliv výrobku lidé hledají hlavně co nejlepší poměr cena/výkon. Na území editorů lze najít poměrně velké množství kvalitních produktů, nicméně ne už tak moc pokud se bavíme o oblasti editorů jazyka VHDL. Celkově editorů VHDL není příliš mnoho a důvod je podlě mě jasný – není taková poptávka. Je jasné, že programátorů, kteří píší např. v Javě, je mnohonásobně více než návrhářů hardwaru používajících jazky VHDL, z malé poptávky tak vyplývá vysoká cena nástrojů.

Pokud se podíváme na editory, které jsou opravdu zadarmo, najdeme asi většinou typické kvalitní editory typu PsPad, gedit, vim atd., tedy hlavně editory, které nám budou

zvýrazňovat syntaxi, ale nějaké širší možnosti pro daný jazyk nenabídnou.

2.2.1 Xilinx ISE

Jeden z nejznámějších produktů pro vývoj hardwaru ve VHDL je monstrózní produkt od firmy Xilinx a to jejich vývojové prostředí Xilinx ISE. Tento program je k dispozici v několika verzích, kde se jejich ceny pohybují od 50 000 do 100 000 korun. To určitě není cena, která by byla příliš přívětivá pro člověka jako jednotlivce. Samozřejmě, že tohle vývojové prostředí nabízí za svoji obrovskou cenu také obrovské možnosti, nicméně otázka zní, zda někdo všechny tyto funkce opravdu potřebuje a využije.

Program umožňuje vygenerování základní struktury VHDL pomocí přehledného okna, kde se zadají údaje jako název entity a architektury nebo se mohou definovat porty, načež program vytvoří soubor s pár řádky a základní strukturou.

Je zde uložena i opravdu velká databáze syntetizovatelných šablon, které je možné využívat a volně vkládat do své práce. Nicméně není možné šablonu upravit jinak, než že ji přepíšete ručně do podoby, která vám vyhovuje.

Xilinx ISE je podle mě velmi kvalitní nástroj, který má opravdu mnoho funkcí, ale působí až příliš robustně a do jednoduchosti sehnání, zprovoznění a používání má daleko.

2.2.2 Sigasi

Další editor VHDL je např. Sigasi od stejnojmenné firmy. První plná verze vznikla teprve před dvěma lety (2010) a dnes se v Evropě prodává za cca 15 000 korun. Je možné stáhnout i tzv. Free starter verzi, která je zadarmo a právě tuto verzi jsem se snažil prozkoumat.

Tento program na první pohled působí o něco čistěji a vypadá docela hezky. Nicméně, program analyzuje text neustále při každém stisku klávesy, tzn. že pokaždé když něco napíšete, už vám vyskakují návrhy na to, co napsat dál. Pokud pak vytváříte nějakou delší konstukci, tak na vás neustále bliká červená kontrolka, která vás upozorňuje na chybu a vy si stále jen říkáte, že opravdu o té nedodělávce víte, ale že prostě potřebujete jen pár sekund na to, abyste začatou konstukci dopsali. Program se také snaží všelijak napomáhat uživateli, obsahuje také databázi šablon, ale ty se týkají spíše jen kratších a často používaných konstrukcí typu if-then-else atd. Během psaní program analyzuje kód a ihned vytváří v navigátoru stromovou strukturu prvků, což je zcela jistě funkce využitelná, protože umožňuje přehledný náhled na celý systém a usnadňuje orientaci a vyhledávání v celém návrhu.

Ač tento nástroj vypadá ze začátku velmi dobře, tak podle mě jsou některé jeho funkce ve výsledku spíše otravné a rušivé. Pokud se bavíme o usnadňování práce uživateli, tak jisté schopnosti má, když např. vylepšuje přístup programu Xilinx ISE v nahrazování identifikátorů ve vloženém kódu.

Kapitola 3

Jazyk VHDL

3.1 Základní informace o jazyce VHDL

VHDL je jazyk z rodiny HDL (hardware descriptipon language) pro popis hardwaru, VHSIC je zkratka pro Very High Speed Integrated Circuits. Tento jazyk je určen pro modelování zejména digitálních systémů a obvodů a je možné ho použít pro tvorbu malých systémů až po systémy velmi rozsáhlé. Není závislý na cílové architektuře, což patří mezi jeho největší přednosti a také umožňuje popis na libovolné úrovni abstrakce, od popisu chování celého systému až po popis na úrovni jednotlivých hradel, ze kterých se systém skládá.[4][6][2][5]

Úplný základ jazyka vznikal v letech 1983 – 1985 jako jedna z částí projektu VHSIC ministerstva obrany Spojených států amerických. Na vývoji se podílely také firmy IBM, Texas Instruments a Intermetrics. Syntaxe vychází z jazyka ADA a je popsána standardem IEEE Std 1076, kde první verzi vytvořila v roce 1987 organizace IEEE. V současnosti je ale nejpoužívanější verze tohoto standardu verze z roku 1993.[4]

3.2 Návrhové jednotky ve VHDL

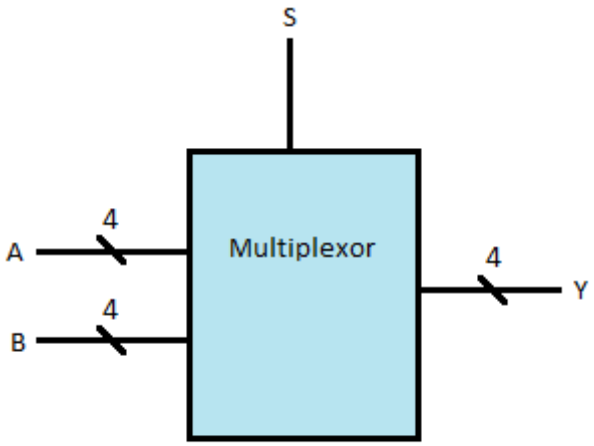
Dva základní prvky pro návrh komponent systému v jazyce VHDL jsou *entita* (ve VHDL – **Entity**) a *architektura* (**Architecture**), kde architektura je sekundární jednotka entity, tedy je na ní závislá. Dále jsou to pak jednotky *konfigurační deklarace* (**configuration declaration**), *knihovní balík* (**package**) a *tělo balíku* (**package body**), které je sekundární jednotkou balíku (**package**).[4]

3.2.1 Entita

Entita obsahuje definici rozhraní komponenty a neříká nic o tom, jak a co daná komponenta dělá, popisuje pouze propojení s ostatními komponentami pomocí signálů. Symbolizuje libovolně velký objekt, může se jednat o jediné hradlo, ale také o rozsáhlý systém obvodů. Každá entita má svoje jedinečné jméno, které je uvedeno za klíčovým slovem **Entity**. Poté může následovat nepovinná sekce **generic**, kde je možné definovat generické konstanty entity, které slouží k parametrizaci dané entity, tedy dávají nám možnost měnit určité konstanty z místa mimo danou entitu. Dále následuje definice všech vstupních i výstupních portů v části **port**. Jednotlivé porty reprezentují vodiče, kde každý má svoje jedinečné jméno, mód přenosu (vstup/výstup apod.) a typ (např. std_logic_vector). [2][4][6]

Příklad dvoukanálového multiplexoru:

```
Entity Multiplexor_1 is
  generic(...);
  port(
    A : in std_logic_vector (3 downto 0);
    B : in std_logic_vector (3 downto 0);
    S : in std_logic;
    Y : out std_logic_vector (3 downto 0);
  );
end Multiplexor_1;
```



Obrázek 3.1: Multiplexor_1

Módy (režimy) přenosu

Směr toku dat můžeme definovat jako: **in**, **out**, **inout**, **buffer** nebo **linkage**.[4]

- **in** – Reprezentuje vstupní port, směr toku dat je zvenku dovnitř systému.
- **out** – Reprezentuje výstupní port, směr toku dat je zevnitř ven ze systému.
- **inout** – Tento port je zároveň vstupní i výstupní, směr toku je tedy možný oběma směry.
- **buffer** – Režim buffer stejně jako režim out reprezentuje výstup, ale s tím rozdílem, že data uvedená na výstup je možné zpetně číst, což v režimu out není možné.
- **linkage** – Tento režim je téměř nepoužívaný.

3.2.2 Architektura

V jednotce architektura je obsažena implementace dané entity, její struktura nebo její chování. Obecně může mít jedna entita více architektur, tedy může být realizována více

způsoby, minimálně však musí být entitě přiřazena alespoň jedna architektura. Každá architektura má opět svoje jedinečné jméno, uvedené za klíčovým slovem **Architecture**, dále následuje klíčové slovo **of** a za ním název entity, se kterou je tato architektura spojena. Dále následuje část pro deklaraci interních signálů a proměnných. Další část obsahuje samotný popis komponenty, popis je nejčastěji behaviorální (popis chování), strukturní nebo tzv. *data-flow* popis. Tato část je umístěna mezi klíčová slova **begin** a **end** a příkazy vložené do této části modelují paralelně prováděné akce. [4][2][6]

Příklad architektury daného multiplexoru popsaný behaviorálně:

```
Architecture Behavioral_1 of Multiplexor_1 is
begin
  proc_1 : process(A, B, S)
  begin
    if (S = "0") then Y <= A;
    else Y <= B;
    end if;
  end process;
end Behavioral_1;
```

Architektura entity Multiplexor_1

3.2.3 Konfigurační deklarace

Konfigurace se používá k propojení entity s architekturou a to v případě, kdy má jedna entita definováno více architektur, pomocí konfigurace je tedy možné jednu z těchto architektur vybrat. Je to nepovinná jednotka a pokud není uvedena, tak se vybere architektura definovaná jako poslední. Definuje se klíčovým slovem **Configuration**. [4][2]

Příklad konfigurační deklarace, která entitě přiřadí jednu ze všech definovaných architektur:

```
Configuration Conf_1 of Multiplexor_1 is
  for Behavioral_4
  end for;
end Conf_1;
```

3.2.4 Knihovní balíky

Knihovní balík umožňuje definovat procedury, funkce, typy nebo operace, které je možné využít k tvorbě popisu. Za klíčovým slovem **Package** jsou deklarovány dané objekty a jejich implementace je poté definována v bloku **Package body**. Mezi nejpoužívanější balíky patří kupříkladu *std_logic_1164*. Tento balík je definován ve standardu IEEE Std 1164-1993 a jsou v něm definovány datové typy *std_logic*, *std_ulogic*, *std_logic_vector* a *std_ulogic_vector*, dále pak základní syntetizovatelné funkce. Další často používané balíky jsou například *std_logic_arith*, *numeric_std* nebo *math_real*.[4][2]

Knihovny a balíky lze poté připojit pomocí klíčových slov **library** a **use**, které se vkládají na úvodní řádky zdrojového souboru.

Připojení konkrétních knihoven a balíků tedy vypadá takto:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.my_package.all;
```

3.3 Typy popisu systému

3.3.1 Behaviorální popis

Behaviorální popis se skládá z klasických konstrukcí imperativního programování, objevují se zde funkce, konstrukce if-then-else, přiřazení atd. Cílem tohoto přístupu je pouze popsat to, jak se bude výsledný systém chovat na základě změn vstupních signálů. Nespecifikujeme tedy skutečné hardwarové provedení, ale pouze popisujeme chování, obdovou realizaci za nás pak řeší syntéza. Problém ovšem může nastat právě v této skutečnosti, proces syntézy může vytvořit systém, který nebude mít takové chování, které jsme očekávali, dokonce ani poté, co v simulaci obvodu vše fungovalo správně. K řešení tohoto problému se dostaneme později. V našem modelovém příkladu multiplexoru se ještě vyskytuje jeden důležitý prvek a to je prvek proces, který slouží právě k behaviorálnímu popisu. [6]

Proces

Proces popisuje chování libovolné části komponenty. Provádí v něm uvedené příkazy sekvenčně, což je rozdíl od paralelního provádění příkazů v sekci **Architecture**. Proces se skládá z jedinečného jména, které můžeme nebo nemusíme uvést, dále pak následuje tzv. *sensitivity list* (seznam citlivých proměnných), který je také nepovinný a poté samotné příkazy popisující sekvenční chování. [1][6]

Proces je uveden klíčovým slovem **process** a poté jsou v závorce uvedeny signály, jejichž změna vyvolá spuštění daného procesu. Poté, co proces celý proběhne, je uveden do stavu čekání na další změnu citlivých signálů. V následujícím modelovém příkladu se tedy proces s názvem *proc_1* spouští při změně vstupních signálů A,B nebo S, samotný popis chování je pak vložen mezi klíčová slova begin a end. Ještě před slovo begin můžeme deklarovat interní proměnné nebo signály. [1][6]

```
proc_1 : process(A, B, S)
  signal Y : std_logic;
begin
    if (S = "0" ) then Y <= A;
    else Y <= B;
    end if;
end process;
```

3.3.2 Strukturní popis

Strukturní popis staví na přesném přepisu schématu realizovaného obvodu. Popisujeme z čeho se daná komponenta skládá, zároveň však využíváme i behaviorálního popisu a to zejména pro popis prvků na nejnižší úrovni. Uvádíme tedy z čeho je komponenta složená, ale musíme také přesně popsat propojení všech dílčích komponent. [6]

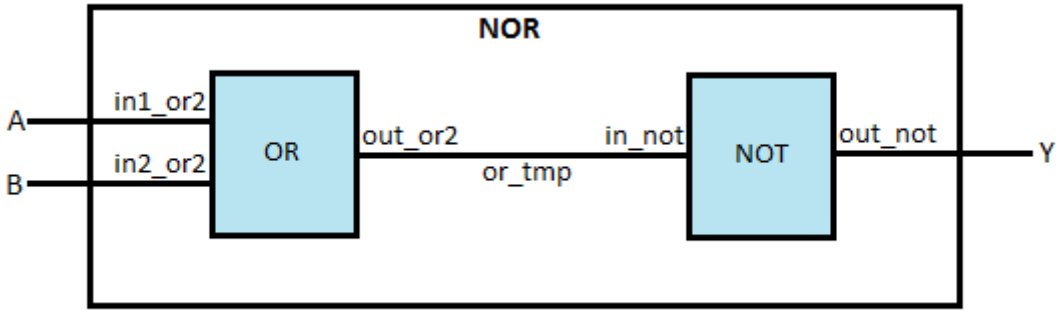
Deklaraci jednotlivých kompoment a deklaraci jejich vstupů a výstupů uvozené klíčovým slovem **Component** vkládáme před klíčové slovo **begin** v sekci **Architecture**. Propojení těchto komponent pak mezi **begin** a **end**.

Příklad strukturního popisu hradla NOR i s definicí rozhraní:

```
Entity NOR_1 is
  port(
    A : in std_logic_vector (3 downto 0);
    B : in std_logic_vector (3 downto 0);
    Y : out std_logic_vector (3 downto 0);
  );
end nor_1;

Architecture struct_1 of NOR_1 is
  Component OR2
  port(
    in1_or2, in2_or2 in std_logic_vector (3 downto 0);
    out_or2 out std_logic_vector (3 downto 0);
  );
  end component;

  Component NOT1
  port(
    input in std_logic_vector (3 downto 0);
    output out std_logic_vector (3 downto 0);
  );
  end Component;

  signal or_tmp : std_logic_vector(3 downto 0);

begin
  or_1 :      OR2 port map(
    in1_or2 => A, in2_or2 => B, out_or2 => or_tmp);

  not_1:      NOT1 port map(
    in_not => or_tmp, out_not => Y);
end struct_1;
```

3.3.3 Popis datových toků (data-flow popis)

V předchozím příkladě je vidět, že sice máme komponenty (*OR2* a *NOT1*) sestavené do určitého systému a že je máme určitým způsobem propojené, nicméně ještě zde chybí definice funkcí jednotlivých komponent. Data-flow popisem modelujeme datové závislosti jednotlivých signálů. Funkčnost těchto dvou komponent se dá tedy popsat jednoduchými rovnicemi s využitím logických operací. [6]



Obrázek 3.2: NOR_1

Doplnění předchozího příkladu o definici funkčnosti *OR2* by tedy vypadalo:

```
Entity OR2 is
  port(
    input1,input2 in std_logic_vector(3 downto 0);
    output out std_logic_vector(3 downto 0);
  );
end OR2;

Architecture dataflow_1 of OR2 is
begin
  output <= input1 or input2;
end dataflow_1;
```

Stejným způsobem by byla poté popsáná i funkčnost komponenty NOT1.

3.4 Některé syntaktické konstrukce VHDL

3.4.1 Signály a proměnné

Signál je prvek, který slouží k přenosu dat a komunikaci mezi komponentami systému¨, reprezentuje tak vlastně fyzický vodič. Deklarace signálu se provádí v definici architektury před klíčovým slovem **begin**. Signály se mohou používat jak uvnitř procesu tak mimo něj. [4]

Syntaxe signálu

```
signal jméno : typ [:= počáteční hodnota];
```

Příklady deklarace

```
signal and1_output : std_logic;                          --(př. 1)
signal and2_output : std_logic_vector (3 downto 0); --(př. 2)
signal and3_output : std_logic_vector (0 to 3);      --(př. 3)
```

Příklady použití

```
and1_output <= and2_output(0);
and2_output <= "0101";
```

V popisu syntaxe vidíme přiřazení do signálu pomocí operátoru `:=`, ovšem tento operátor se používá pouze pro přiřazení počáteční hodnoty signálu, jinak se používá operátor `<=`. V předchozích příkladech jsou uvedeny tři možnosti deklarace signálu. V př. 1 je signál deklarován jako jeden vodič. V př. 2 a př. 3 jsou deklarovány signály jako svazek vodičů, tedy sběrnice. Rozdíl je pouze ve významnosti bitů, v př. 2 je nejvýznamější bit č. 3 a nejméně významný je bit č. 0. U př. 3 je naopak, tedy nejvýznamější bit je 0. bit.

Jak jsme si již řekli, příkazy uvedené v procesu se provádí sekvenčně, nicméně není to tak u přiřazování hodnot do signálu. Nastavení signálu se provede až v době, kdy všechny procesy doběhnou do konce. Pokud tedy v jednom procesu chceme několikrát vložit hodnotu do signálu, tak se provede pouze to poslední a ostatní budou ignorovány. Tímto přístupem je zajištěno, že nebude záležet na pořadí procesů a je tím tedy simulováno paralelní provádění.[6]

Pokud si potřebujeme ukládat nějakou hodnotu mezivýpočtu, použijeme pro to proměnnou. Do proměnné je možné vkládat hodnoty během provádění procesu a toto vložení se provede ihned. Deklarace proměnné je možná pouze uvnitř konkrétního procesu, stějně tak ji mimo proces nemůžeme použít. Výjimkou je ovšem sdílená proměnná (shared variable), která musí být deklarována mimo proces.[4]

Syntaxe proměnné

```
[shared] variable jméno : typ [:= počáteční hodnota];
```

Příklady deklarace

```
variable citac : integer range 0 to 255 := 0;
shared variable tmp : boolean;
```

Příklad použití

```
citac := citac + 1;
```

Kapitola 4

Popis hardwaru ve VHDL

4.1 Nejčastěji používané komponenty

V této části se jenom lehce zmíním o funkčních blocích, které můžeme nazvat jako základní, protože se používají asi nejčastěji. Zde je popis vybraných komponent. Podrobnější informace lze najít v [4].

Dekodér

Dekodér je blok, který převádí binární kód na *kód 1zN*[1]. To znamená, že je v daném čase aktivní vždy jenom jeden z N výstupů. Často dekodéry obsahují ještě vstup, který zablokuje všechny výstupy a uvede je do neaktivního stavu. Funkce dekodérů je vybírat jeden z N obvodů a ve VHDL se pro to dá použít například konstrukce **case**, **if-then-else** atd.

Kodér

Kóder má opačnou funkci než dekodér, když převádí kód 1zN na binární kód, takže na vstupu kodéru musí být v jeden okamžik vždy aktivní jen jeden signál. Aby toho bylo dosaženo, tak se zpravidla používá tzv. *prioritní kodér*, který dovoluje, aby byl aktivní pouze jeden signál na vstupu.

Multiplexor

Multiplexor má funkci přepínače mezi signály nebo mezi skupinami signálů. Na vstup multiplexoru jsou přivedeny vstupy adresy a vstupy signálů, které ma multiplexor přepínat. Například dvoukanálový multiplexor, který se často používá, obsahuje jeden vstup adresy, kterým lze adresovat právě dva vstupy multiplexoru. Podobně čtyřkanálový multiplexor obsahuje čtyři vstupy signálů, ale potřebuje dva vstupy adres atd. I zde se používá nejčastěji konstrukce pomocí příkazu **case**.

Demultiplexor

Demultiplexor má naopak jeden kanál vstupní a více výstupních a pomocí přepínacího signálu je vybrán jeden signál na výstupu, na který bude přiveden signál ze vstupu. Podobně jako u multiplexoru má demultiplexor takový počet vstupů adresy, aby bylo možné adresovat všechny výstupy demultiplexoru. Tedy například šestnáctikanálový demultiplexor

[1] Jeden z N bitů je jiný než ostatní (např. 11110111)

má šestnáct výstupů, čtyři vstupy adresy a jeden vstupní signál. Znovu se dá popisovat příkazem typu **case**.

Datový registr

Datový registr je obvod složený z klopných obvodů typu D, které mají společný hodinový signál a vzorkují vstupní signál při vzestupné popř. sestupné hraně tohoto hodinového signálu (obvody řízené hranou). Nebo mohou vzorkovat vstupní signál po celou dobu, kdy je hodinový signál v log. 1 (obvody řízené úrovní). Registry mohou také obsahovat nulovací vstup, poté mluvíme o registrech buď se *synchronním*, nebo s *asynchronním nulováním*.

Posuvný registr

Posuvný registr je tvořen skupinou klopných obvodů typu D, které jsou spojené do kaskády a mají společný hodinový signál. Pokud uvažujeme posuvný registr s posunem vpravo, tak tento registr je doplňován daty zleva a data vpravo se ztrácí. Obraceně funguje posuvný registr s posunem vlevo. Existuje i možnost posunu o více bitů, který se zajišťuje pomocí vhodně zapojených multiplexorů, díky kterým je možné přepínat nejen zleva nebo zprava sousedící klopné obvody.

Pokud se některý z výstupů posuvného registru připojí na seriový vstup, tak vzniká posuvný registr se zpětnou vazbou. Takto je například realizován tzv. *kruhový registr*, který má na vstup připojen výstup z posledního klopného obvodu. Data, která jsou do tohoto registru vložena, pak rotují s každým hodinovým signálem o jedno místo.

Čítač

Další velmi často používanou skupinou komponent jsou čítače. Všechny mají shodné to, že počítají impulzy. Čítač na každý impulz přejde do jiného stavu a prochází tak předem daný počet stavů. Používají se čítače, které mohou počítat směrem nahoru nebo dolů, kde se na jeden impulz většinou mění více bitů[2]. Potom ale existují i čítače, které počítají v tzv. *Grayově kódu*, kde se vždy mění pouze jeden bit. Toto byly čítače počítající buď nahoru, nebo dolů, avšak jsou i čítače tzv. *reverzivní*, které mohou čítat oběma směry. A nakonec ještě můžeme rozdělit čítače na tzv. *synchronní* a *asynchronní*, kde u synchronního se všechny klopné obvody překlápí ve stejný okamžik, kdežto u asynchronního ne.

4.2 Syntéza a problémy s ní spojené

Syntéza je automatický proces, který se skládá z několika úrovní. Převádí se při něm systém popsaný v jazyce (V)HDL na strukturu na úrovni hradel (tzv. *netlist*).[2]

Vývojář může vytvářet model na libovolné úrovni abstrakce, a proto může být proces transformace na strukturu hradel nelehký úkol. Zároveň je důležité si uvědomit, že programování v jazyce VHDL se od programování v jazycích typu C, Java apod. značně liší, i když to tak na první pohled nemusí vypadat, protože se používají podobné konstrukce. Tedy pokud mluvíme o tom, že chceme skutečně z našeho popisu vytvářet hardwarovou realizaci. Pokud nám jde pouze o simulaci daného obvodu, máme ruce poněkud volnější, ale v případě, že tvoříme model pro syntézu, musí být námi vytvořené konstrukce tzv. *syntetizovatelné*. Jinak řečeno musíme tvořit takové konstrukce, které je možné hardwarově

[2]Př. Přechod z čísla 7 na 8. Změní se více bitů najednou, zde dokonce všechny (0111 na 1000).

realizovat. V hardwaru všechny prvky pracují paralelně, což je právě rozdíl od sekvenčního zpracování programu procesorem (jako např. při psaní v C, Javě atd.). Toto může být omezující, protože speciálně jazyk VHDL je velice mocný, ale pro účely syntézy můžeme využít jenom některé konstrukce. Nicméně na trhu v současné době můžeme najít syntetizátory od různých tvůrců, které se občas líší v tom, co dokáží syntetizovat a zároveň se obecně úroveň syntetizátorů zvyšuje, takže zvládají stále složitější konstrukce. [7] [4]

4.3 Syntetizovatelné šablony

V kapitole 4.2 už bylo zmíněno, že pokud chceme vytvářet systémy, které se budou poté hardwarově realizovat, musíme používat konstrukce jazyka, které jsou syntetizovatelné. Pro tyto účely se používají tzv. *syntetizovatelné šablony*, u kterých máme jistotu, že je syntetizátor správně pochopí a výsledný obvod se bude chovat podle očekávání.[7]

4.3.1 Příklady syntetizovatelných šablon

Zde uvedu několik příkladů syntetizovatelných šablon pro vybrané komponenty. Popis funkčnosti a použití těchto komponent byl uveden v kapitole 4.1, proto odkazuji na příslušnou část. Za všechny jsem vybral čítač a multiplexor, ale podrobnější informace je možné najít v [7].

Čítač

Zde je příklad synchronního čítače se synchronním nulováním, který čítá do hodnoty dané počtem bitů signálu ADDR, tedy do $(2^8 - 1)$. Poté, co dosáhne této hodnoty, přeteče a počítá znovu od nuly.[7]

```
architecture beh of citac is
  signal cnt : std_logic_vector(3 downto 0);
  signal ADDR : std_logic_vector(7 downto 0);
begin

  ADDR <= cnt;

  process(CLK)
  begin
     if (CLK'event) and (CLK='1') then
        if (RST='1') then
           cnt <= (others => '0');
        elsif (EN='1') then
           cnt <= cnt + 1;
        end if;
     end if;
  end process;

end architecture;
```

Multiplexor

Toto je ukázka multiplexoru 4 na 1, který vybírá jeden ze čtyř signálů a který je ve VHDL popsán pomocí konstrukce **if-elsif**.[7]

```
process(A,B,C,D,SEL)
begin
    if (SEL="00") then
        Y <= A;
    elsif (SEL="01") then
        Y <= B;
    elsif (SEL="10") then
        Y <= C;
    else
        Y <= D;
    end if;
end process;
```

Kapitola 5

Návrh řešení

Nyní je potřeba zvolit si jedno z možných řešení. První možností je implementace vlastního řešení do některého z již existujících editorů. Výhod v tomto směru nacházím několik. Hlavní výhoda je podle mého názoru ta, že se většinou můžeme spolehnout na kvalitně navržené uživatelské rozhrání, zároveň je toto rozhraní již otestováno a neměly by se v něm vyskytovat žádné výraznější chyby. V editoru také často vidíme možnost si uživatelské rozhraní přesouváním různých bloků variabilně měnit podle svých představ. Další výhodu vidím v tom, že hotový editor většinou nabízí všechny základní funkce, které uživatelé vyžadují a potřebují. Tím myslím zejména funkce typu zvýrazňování syntaxe vybraného jazyka, automatické doplňování klíčových slov a dalších syntaktických konstrukcí, rozsáhlé možnosti nastavení editoru jako třeba nastavení barev, způsoby odsazovaní zdrojového kódu a spoustu dalších. Nicméně vedle výhod zde spatřuji i řadu nevýhod. Za tu největší považuji práci v cizím prostředí. Stávající editor nám jistě neumožní implementovat všechno, co bychom si přáli a co bychom si mohli dovolit pokud píšeme aplikaci od základu sami.

Já jsem se rozhodl napsat aplikaci sám, abych mohl použít většinu svých vlastních nápadů na realizaci. Nicméně využívám služeb vývojového prostředí NetBeans IDE[1], které nabízí značné usnadnění tvorby uživatelského rozhraní. Prostředí NetBeans poskytuje možnost vytvořit uživatelské rozhraní velmi jednoduše a intuitivně, načež generuje do zdrojového kódu přibližně 200 řádků, které obsahují zejména inicializace objektů a jejich umisťování do daného okna[2].

5.1 Zvolený programovací jazyk

Jako programovací jazyk jsem zvolil objektově orientovaný jazyk Java, který je podle mě pro podobný projekt velice vhodný. Pro tvorbu uživatelského rozhraní obsahuje Java grafickou knihovnu JFC (Java Foundation Classes), též známou jako knihovna Swing, která byla použita pro tento projekt.[3]

Jazyk Java vyvíjela firma Sun Microsystems od roku 1991 (tehdy se ještě jmenoval Oak), kde stavěla na základech jazyku C a C++ a oficiálně byla představena až v roce 1995. Velmi rychle vzbudila zájem průmyslu, hlavně kvůli tomu, že byl právě Internet a WWW na vzestupu a právě v Javě byl viděn pro tyto účely velký potenciál.[3]

Jedna z těch nějvětších výhod Javy je její multiplatformovost. Programy psané v Javě

[1] Při vývoji byla použita verze NetBeans IDE 6.9.1, volně dostupná na `http://www.netbeans.org`.

[2] Tyto části zdrojového kódu jsou označeny jako nepůvodní a jsou to jediné části aplikace, které jsem nenapsal sám

jsou přenositelné mezi různými operačními systémy bez nutnosti překladu na daném systému. Přenositelnosti je dosahováno pomocí tzv. *bajtkódu*. Programy psané v Javě nejsou překládány do strojového jazyka počítače, ale právě do již zmiňovaného bajtkódu, který je nezávislý na cílové platformě, takže programátor nemusí dopředu vědět, na jakém počítači se bude program spouštět. Bajtkód je poté interpretován pomocí speciálních programů Javy, které se nazývají *Java platforma*. [3]

Java platformu tvoří dvě hlavní části. Za prvé je to tzv. *virtuální stroj* (*JVM – Java Virtual Machine*) a za druhé je to *Java Core API*. Java Core API je aplikační programové rozhraní, to obsahuje velký počet knihovních tříd, které se vyskytují v každém prostředí, kde se Java používá. Toto je věc, kterou ocení snad všichni programátoři, kteří píší v Javě, protože mohou velmi často využívat služeb API a neřešit již vyřešené problémy.[3]

Nevýhody interpretovaných jazyků můžeme naopak vidět v tom, že oproti kompilovaným jazykům typu C jsou poměrně pomalé. To je způsobeno odlišnou vnitřní filosofií, kde Java zakládá na tom, aby byla pohodlná pro programátora, ale pro počítač je značně výpočetně náročná. V Javě existuje řešení v podobě tzv. *Just in Time kompilátorů* (JIT), které zajišťují alespoň částečné zvýšení rychlosti na úroveň programů napsaných například ve zmiňovaném jazyce C.[3]

Důvody, díky kterým jsem se rozhodl pro Javu, jsou hlavně pohodlnost používání, kde se programátor například nemusí starat o uvolňování paměti (vše se děje automaticky, zajišťuje to tzv. *garbage collector*), dále má obrovské možnosti využívání knihovních tříd v Java Core API, má k dispozici grafickou knihovnu JFC (Swing) pro vytváření uživatelských prostředí a v neposlední řadě i mnoho velmi mocných vývojových prostředí, jako v mém případě třeba zmiňované NetBeans IDE.

5.2 Návrh uživatelského rozhraní

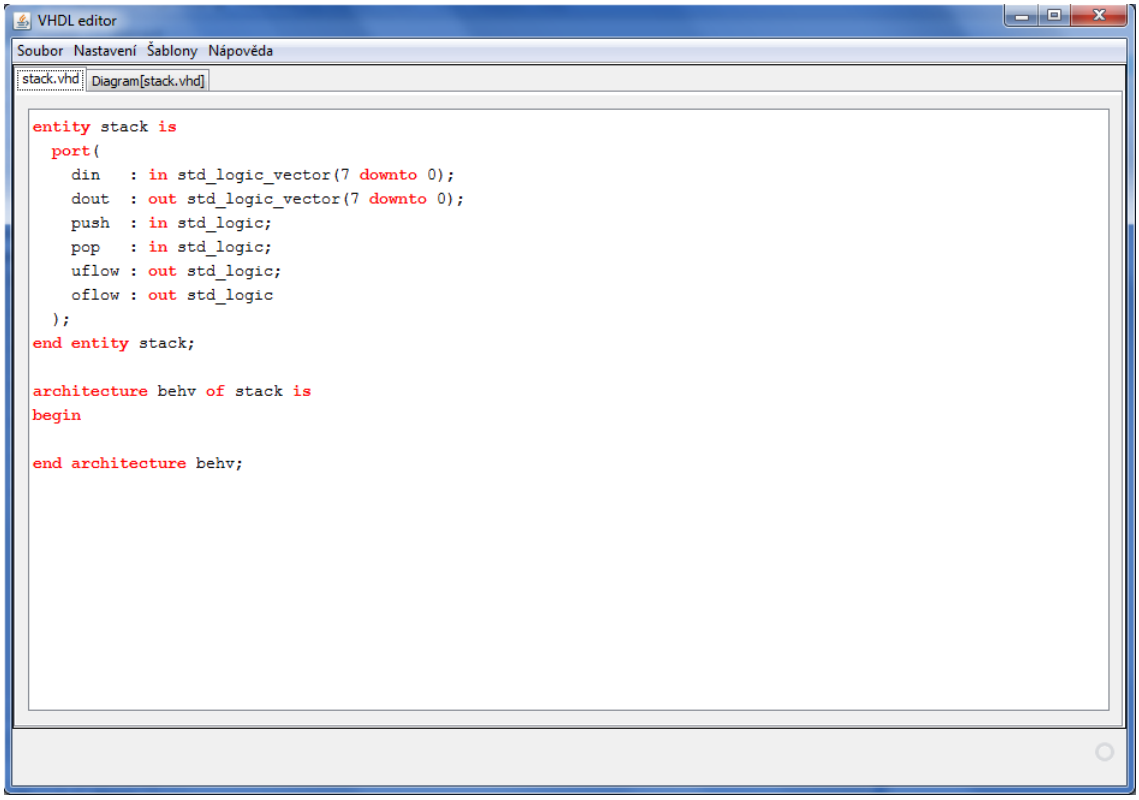
Uživatelské rozhraní jakéhokoliv editoru musí být především přehledné a uživatelsky přívětivé. Uživateli musí být po krátké době jasné, jak s daným programem pracovat, musí pochopit, jaké má možnosti a co naopak editor nenabízí. Není příliš vhodné narušovat běžně známe a ověřené modely uživatelského rozhraní, protože to uživatele zbytečně nutí měnit svoje zvyklosti. Podle mého názoru není intuitivnějšího přístupu než možnosti tvorby návrhu v grafickém prostředí. Velmi lehce jsem se inspiroval právě vývojovým prostředím typu NetBeans a vytvořil program jako tzv. klikátko, i když samozřejmě tento program nemá ani zdaleka takové schopnosti.

Hlavní okno programu obsahuje dvě hlavní části, jsou to textová část a grafická část. Je možné mezi nimi jednoduše přepínat, kdy může být zobrazena pouze jedna z těchto částí, ne obě najednou. Funkcionalita je ovšem propojená. Kromě těchto dvou hlavních oblastí hlavní okno obsahuje menu, ve kterém jsou položky *Soubor*, *Nastavení*, *Šablony* a *Nápověda*.

5.2.1 Textová část

Textová část je zobrazena na obrázku č. 5.1. Je tvořena textovým polem pro zdrojový kód, nic víc zde nenajdeme. Program obsahuje funkci zvýrazňování syntaxe, konkrétně zvýrazňování klíčových slov jazyka VHDL a komentářů. Uživatel má možnost vlastního nastavení barvy pozadí, textu i barev zvýrazňovaných prvků v kódu.

Program nabízí vytvoření základní struktury VHDL entity pomocí dialogového okna, které se otevře po zmáčknutí na položku v menu *Soubor -> Nový*. Toto okno (obrázek č. 5.2) má několik záložek – v záložce *Základní* je možné zadat jméno entity a jméno architektury,



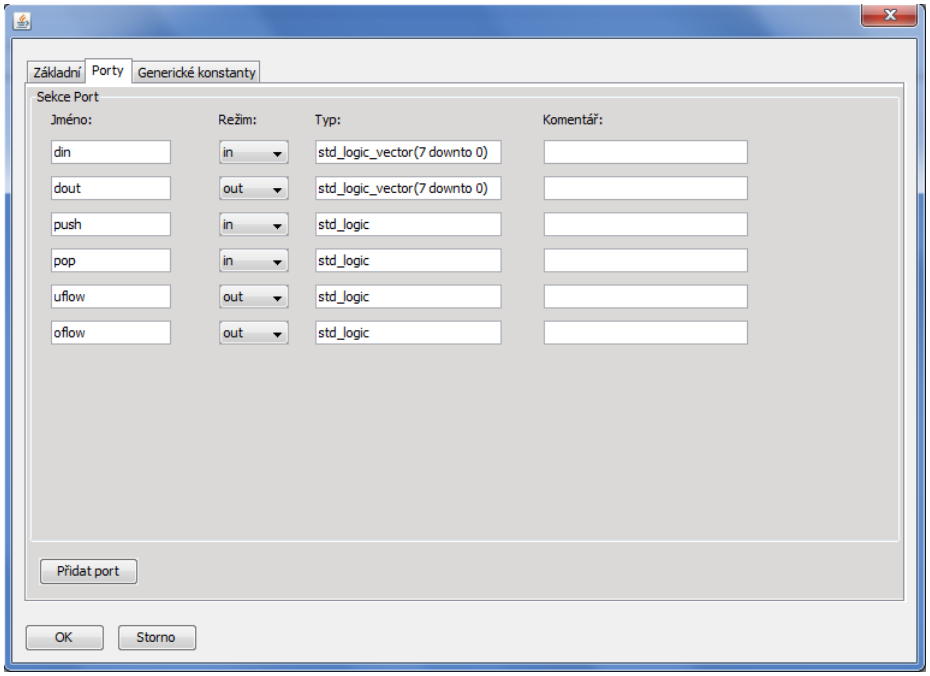
Obrázek 5.1: Textová část hlavního okna – vygenerovaná základní struktura

dále pak připojit knihovny a zpřístupnit knihovní balíky. Další záložky jsou označeny jako *Porty* a *Generické konstanty*, kde může uživatel, jak už z názvů vyplývá, přidávat porty a generické konstanty. Způsob zadávaní je u obou téměř stejný, u portu se zadává jméno, režim přenosu, datový typ a poznámka, která se poté vygeneruje jako komentář za definici daného portu, u generických konstant je to to samé s výjimkou režimu přenosu. Po stisknutí tlačítka *OK* se do zdrojového kódu vygeneruje základní struktura VHDL entity, která je vidět na obrázku č. 5.1.

5.2.2 Grafická část

Obrázek č. 5.3 zobrazuje grafickou část editoru. Obsahuje panel nástrojů pro ovládání diagramu a panel pro návrh a manipulaci s diagramem. V panelu nástrojů jsou tlačítka *Vytvořit komponentu*, *Vytvořit signál* a *Napojit na signál/port*. Práce s diagramem je velmi jednoduchá, uživatel má možnost pouze základních operací, které vybírá pomocí těchto tří tlačítek v panelu nástrojů.

Samotný diagram je pak tvořen komponentami s podobným vzhledem, které jsou propojeny za pomocí signálů, kde každá komponenta je zobrazena jako obdelník s nastavitelnou velikostí. Dále jsem se rozhodoval, zda se budou zobrazovat i jednotlivé porty každé komponenty, anebo signál graficky povede kamkoliv do komponenty. Nakonec jsem zvolil zobrazení portů v podobě malých čtverečků, se kterými se ale nedá v rámci komponenty manuálně hýbat. Jejich pozice vzhledem ke komponentě je tak stále stejná. S celými komponentami se samozřejmě hýbat dá a signály na ně napojené při pohybu komponenty svoje pozice mění rovněž.



Obrázek 5.2: Vytvoření nového souboru – záložka *Porty*

Každý signál je pak tvořen z úseček a ze zalamovacích bodů. Signály jsou pohyblivé pouze v zalamovacích bodech, ale je možné je propojovat i mimo ně, tedy na kterékoliv úsečce signálu. Mechanismus tvorby signálu nedovoluje uživateli vytvářet jiné, než pravoúhlé tvary signálu a dělá tak vzhled návrhu na pohled přehlednější. Původně měl tento mechanismus udržovat pravé úhly mezi úsečkami i při každém pohybu s kterýmkoliv prvkem v diagramu, nicméně při složitějších návrzích nebyl výsledek příliš uspokojivý, takže jsem od toho přístupu raději upustil a nyní to tedy funguje tak, že při pohybu komponenty se mění pozice pouze komponentě nejbližší úsečky signálu.

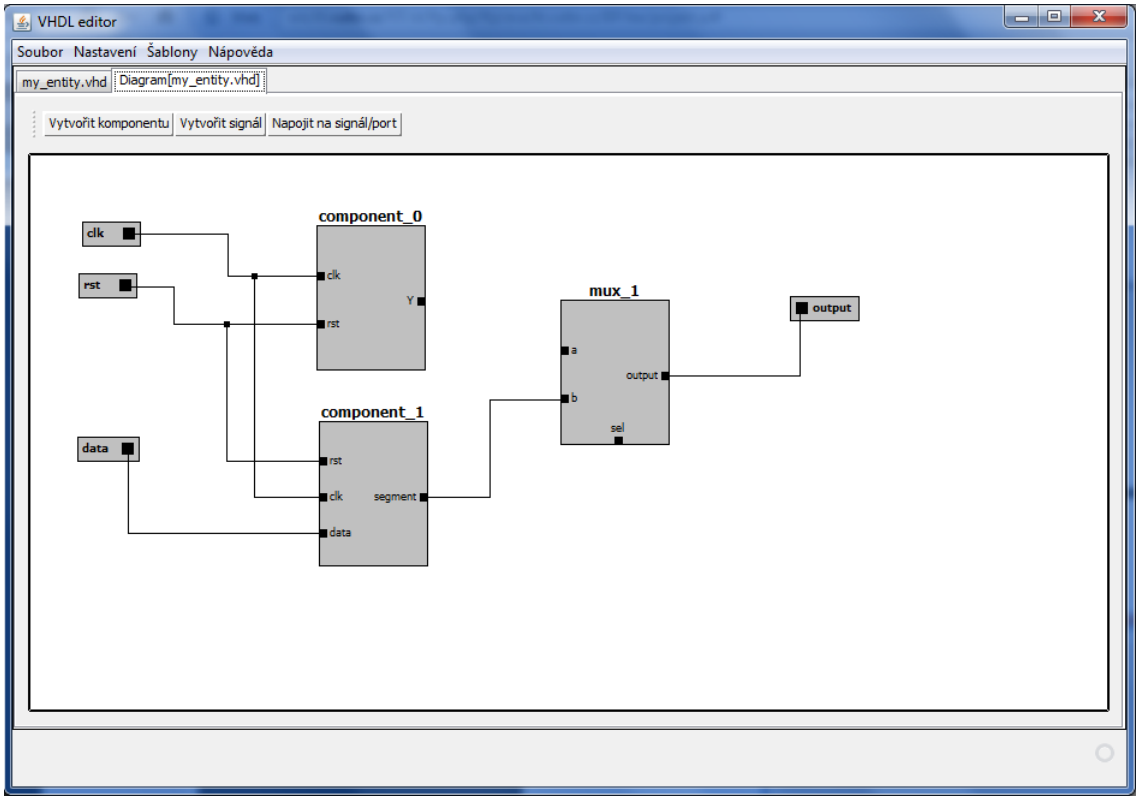
Různé je zobrazování popisků jednotlivých prvků. Při návrhu jsem se snažil hlavně o to, aby toho uživatel zároveň viděl co nejvíce, ale nechtěl jsem, aby se mu v návrhu pletlo příliš mnoho popisků. Permanentně zobrazeny jsou tedy pouze jména portů a jména komponent, ale jejich detailnější popis, stejně jako popis signálu se zobrazí až po najetí myší nad daný prvek.

5.2.3 Další okna

Program ještě obsahuje dvě velmi podstatná dialogová okna, která jsou při práci s programem často využívána. První je okno pro definici a editaci komponenty a druhé pro vkládání a zapojování syntetizovatelných šablon.

Vytvoření komponenty

Toto okno je zobrazeno na obrázku č. 5.4, kde je vidět, že se ještě skládá ze dvou záložek. Uživatel si v tomto okně definuje nové komponenty, nebo využívá komponent, které si už vytvořil a uložil dříve. Každé komponentě se zadá jedinečné jméno v rámci entity a poté je možné buď zatrhnout políčko *Vlastní* a nadefinovat si novou komponentu, nebo ze stromu komponent jednu vybrat.



Obrázek 5.3: Grafická část hlavního okna v průběh tvorby schématu

Na obrázku č. 5.5 je vidět práce s komponentou. V záložce *Porty* je možné nadefinovat porty komponenty a libovolně zvolit její vzhled pomocí intuitivního panelu. Uživatel si může takto nadefinovanou komponentu uložit a později znovu nahrát. Uložení komponenty probíhá tak, že se analyzují všechna editační políčka v okně a hodnoty se ukládají v podobě zakodovaného textového řetězce[3] do souboru, kde název tohoto souboru odpovídá typu dané komponenty.

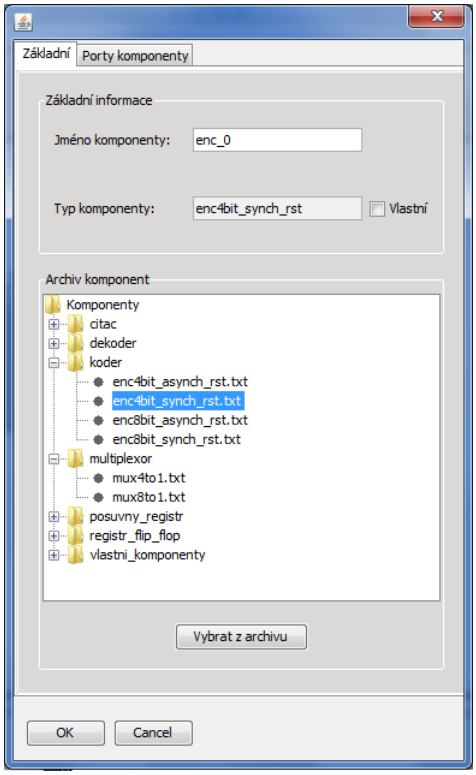
Vkládaní šablony

Na obrázku č. 5.6 je vidět okno, které slouží k výběru a zapojování syntetizovatelných šablon. Okno je logicky rozděleno na několik oblastí.

Na levé straně se nachází textové pole, které zobrazuje zdrojový kód aktuálně vytvářené šablony. Zde je šablona pro uživatele již upravena tak, aby se v ní mohl lehce orientovat a je tedy zbavena všech kódovacích znaků, podle kterých se orientuje analyzátor.

V okně je dále vpravo nahoře oblast nazvaná *Entita* a jsou zde vloženy dva seznamy signálů. První z nich obsahuje všechny signály a porty, které se v entitě vyskytují. Druhý potom pouze ty signály, které jsou připojeny na komponentu, pro kterou danou šablonu tvoříme. Zde je nutno ještě podotknout, že toto okno se dá zobrazit dvěma způsoby. Uživatel může pomocí tlačítek *Šablony -> Vložit šablonu* v hlavním menu programu spustit toto okno s tím, že nechce šablonu přiřazovat konkrétní komponentě, kterou v diagramu vytvořil. V tomto případě zůstane seznam *Signály napojené na komponentu* prázdný. V druhém případě uživatel klikne pravým tlačítkem na některou z komponent v grafické části ve vyskakovacím

[3]Toto téma téma bude více rozebíráno v kapitole č. 5.3



Obrázek 5.4: Výběr komponenty z archivu

okně zvolí rovněž možnost *Vložit šablonu*. Toto řešení jsem zvolil z toho důvodu, že v entitě se mohou vyskytovat desítky signálů nebo portů a uživatel stejně hledá jenom zlomek z nich. Tento způsob mu to značně usnadní.

Poslední panel s názvem *Šablona* obsahuje stromový archiv šablon a dále pak seznam signálů, které se nachází v šabloně. Archiv modeluje strukturu stejného adresáře jako archiv komponent v okně *Vytvořit komponentu*[4]. Seznam *Signály šablony*, jak už název napovídá, zobrazuje všechny signály, které se vyskytují v aktuálně zpracovávané šabloně. Všechny tyto signály může uživatel postupně nahradit některým ze signálů entity. Udělá to tak, že v seznamu vybere příslušné položky a stiskem tlačítka *Napojit signál na šablonu* se signály prohodí. Uživateli také napomáhá automatické zvýrazňování signálu šablony, který má v daný okamžik vybraný. Toto výrazně zlepšuje orientaci ve zdrojovém kódu šablony.

5.3 Forma šablony a popisu komponenty

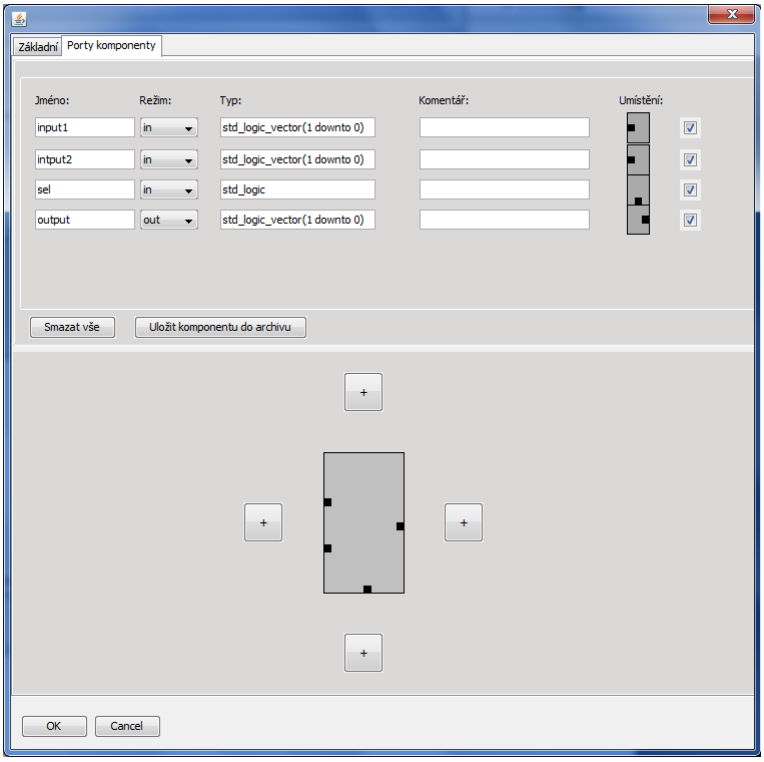
V této kapitole bude ukázáno, jakým způsobem jsou zakódovány syntetizovatelné šablony a jak jsou popsány uložené komponenty. Navrhl jsem strukturu, která tyto dvě věci spojuje dohromady. Je tedy možné uchovávat popis portů a vzhledu komponenty[5] spolu s popisem jejího chování, ale není to vyžadováno, je to pouze možnost pro uživatele.

Každý soubor je složen ze dvou sekcí:

1. Sekce pro definici portů komponenty

[4] Vysvětlení se věnuje kapitola č. 5.3.

[5] Vzhledem je zde myšleno, jakým způsobem se bude v grafickém editoru zobrazovat.



Obrázek 5.5: Definice komponenty

2. Sekce pro šablonu chování

Sekce pro definici portů komponenty

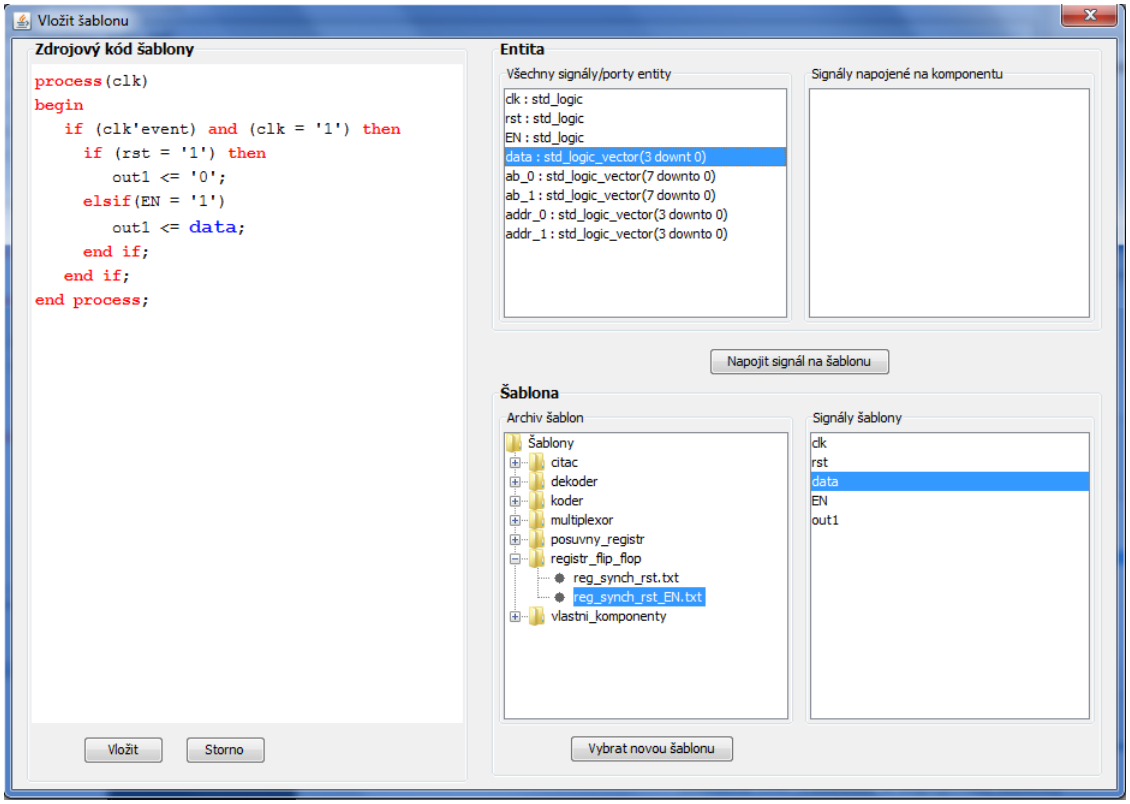
Pokud má uživatel uloženou nějakou svoji komponentu, je potřeba, aby si o ní program pamatoval následující údaje:

- Typ komponenty (odpovídá názvu souboru)
- Názvy všech portů
- Režimy přenosu portů
- Datové typy portů
- Vzhled portů (kde budou jednotlivé porty na komponentě umístěny)

S tím, že povinné jsou všechny položky kromě režimů přenosu a datových typů portu. Ostatní jsou k úspešnému vykreslení komponenty nutné. Ještě vysvětlím pojem *typ komponenty*. Je to název, který by se dal považovat za název třídy komponent, jedinečný identifikátor uživatel komponentě (tedy instanci třídy) přidělí až při jejím vytváření. To bylo popsáno v kapitole č. 5.2.3 a zobrazeno na obrázku č. 5.4.

Každá položka má svoji jedinečnou značku uvedenou za symbolem '$'. Konkrétní hodnota je pak uvedena hned vzápětí ve složených závorkách a jednotlivé položky jsou oddělovány středníkem. V následující úkazce je syntaxe popisu jednoho portu.

```
$s{...};$m{...};$t{...};$v{1-4};$,
```



Obrázek 5.6: Vkládání a zapojování šablony

kde je za `$s` (z angl. *signal*) uveden název, za `$m` (z angl. *mode*) je režim přenosu, za `$t` (z angl. *type*) je datový typ a poslední položka je `$v` (z angl. *visualization*), kde je uvedena strana, kde se port vykreslí (čísluje se od 1 do 4 po řadě levá, horní, pravá a spodní strana). Každý tento řádek tak popisuje jeden z N portů a celá sekce je ohraničena klíčovými slovy `$$defbegin` a `$$defend`.

Příklad definice komponenty z obrázku č. 5.5:

```
$$defbegin
  $s{input1};$m{in};  $t{std_logic_vector(1 downt 0)};$v{1};$
  $s{input2};$m{in};  $t{std_logic_vector(1 downt 0)};$v{1};$
  $s{sel};    $m{in};  $t{std_logic};                  $v{4};$
  $s{output};$m{out};$t{std_logic_vector(1 downt 0)};$v{3};$
$$defend
```

Sekce pro šablonu chování

Každá šablona musí obsahovat i sekci pro definici portů, která je ale pro použití šablony spíše výčtem signálů. To znamená, že jde zde nutné uvést všechny signály, které se budou v šabloně používat. Není ovšem nutné uvádět v ní všechny parametry každého signálu, povinný je jenom jeho název, protože pro použití šablony jsou ostatní bezpředmětné. Definice šablony se uvádí za klíčové slovo `$$templatebegin`. Tělo šablony se kóduje tak, že se každý

signál ve zdrojovém kódu označí opět symbolem `$s`, do složené závorky se uvede název signálu a ještě se přidá druhá závorka, která se nechá prázdná a kam poté uživatel vloží svůj vlastní název signálu nebo portu.

Příklad šablony chování multiplexoru:

```
$$templatebegin
process($s{A}{}, $s{B}{}, $s{C}{}, $s{D}{}, $s{S}{})
begin
  $s{Y}{} <= $s{D}{};
  if ($s{S}{} = "") then
      $s{Y}{} <= $s{A}{};
  elsif ($s{S}{} = "") then
      $s{Y}{} <= $s{B}{};
  elsif ($s{S}{} = "") then
      $s{Y}{} <= $s{C}{};
  end if;
end process;
```

Celý soubor má následující strukturu:

```
$$defbegin
 ...
-- definice signálů nebo portů
 ...
$$defend

$$templatebegin
 ...
-- tělo šablony
 ...
```

5.4 Analyzátor a generátor zdrojového kódu

Pokud má být jakýkoliv editor schopný uživateli smysluplně usnadňovat práci, musí také umět poznat, co píše a část práce za něj dodělat. Nabízí se otázka, jak daleko až zajít, aby to bylo skutečně prospěšné a ne, aby se zanášely do práce návrháře chyby nebo obrovské úseky kódu, které stejně musí potom sám upravovat a přepisovat. Určitě vhodnými pomůckami, které potřebují analyzátor kódu, jsou funkce automatického doplňování klíčových slov nebo nabízení použitých identifikátoru prvků v závislosti na konkrétním kontextu atp. V tomto programu ale nic takového implementováno není, vytvořil jsem pouze jednoduchý analyzátor, který se mj. stará o zvýrazňování syntaxe.

5.4.1 Analýza zdrojového kódu

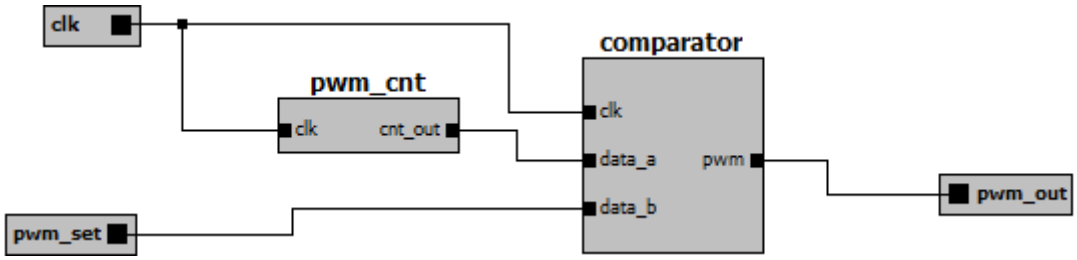
Návrh systému se tvoří hlavně v grafickém prostředí, takže se struktura jednotlivých prvků VHDL ukládá na základě toho. Proto není nutné, aby program obsahoval nějaký obzvláště

kvalitní lexikální nebo syntaktický analyzátor. Program si vystačí pouze z několika metodami, které analyzují a hledají pouze vybrané konstrukce nebo různé delší úseky zdrojového kódu. Analyzátor je tedy schopný vyhledat např. deklarační část architektury nebo deklaraci entity, odkud pak vytváří záznamy o všech portech a signálech entity. Dále umí najít informace o tom, na kterých indexech jednotlivé části začínají a končí, které později využívá k tomu, aby se generovaný kód vkládal na správná místa. Ostatní údaje jako třeba informace o komponentách si program ukládá výhradně na základě grafického modelu systému vytvořeného uživatelem.

5.4.2 Generování zdrojového kódu

Generování kódu je plně iniciováno uživatelem a nikdy neprobíhá bez jeho požadavku. Generovaný kód vzniká na základě údajů, které uživatel buď vyplní do daných editačních políček, nebo na základě vytvořeného modelu systému. Generátor má přístupné informace o základních údajích komponent a o jejich propojení, dále vychází z informací od analyzátoru, který mu poskytuje indexy, kam má daný kód vložit. Program umí při vytvoření nového souboru vygenerovat základní strukturu VHDL, toto je zobrazeno na obrázku č. 5.2. Dále pak generuje deklarační část komponenty (příkaz **component**) a také použití komponenty (příkaz **component port map**), kde na sebe mapuje jednotlivé porty komponenty s vnitřními signály entity nebo porty této entity.

Ukázka generovaného kódu podle nakresleného schématu[6]:



Obrázek 5.7: Schéma vytvořené v grafickém editoru programu

```
...
component pwm_cnt
  port(
    clk     : in std_logic;
    cnt_out : out std_logic_vector(7 downto 0));
end component;
...
pwm_cnt_0 : pwm_cnt
  port map(
    clk => clk,
    cnt_out => cnt_out);
...
```

[6]Schéma bylo překresleno z [5].

Kapitola 6

Realizace aplikace

Tato kapitola se zabývá popisem realizace těch nejdůležitějších částí programu. Celá práce vznikala, jak už bylo řečeno dříve, v prostředí NetBeans IDE, které ulehčuje vývoj nejen uživatelského rozhraní aplikace.

6.1 Uživatelské rozhraní

Uživatelské rozhraní bych rozdělil v tomto programu na dvě části, to první je panel pro grafický návrh systému a druhou jsou všechny ostatní dialogová okna včetně hlavního okna. Ve vývojovém prostředí je možné vytvářet velmi lehce různá okna a vkládat do nich komponenty s tím, že poté toto prostředí automaticky generuje zdrojový kód, který obsahuje deklaraci a inicializaci grafických komponent, poté nastavení základních vlastností a nakonec umístění do tzv. *layoutu* okna. Všechen tento kód je obsažen v metodě `initComponents()`, kterou obsahuje každá třída popisující některé z oken programu.

Hlavní okno představuje třída `EditorMainMWindow`, která rozšiřuje třídu `FrameView`. V tomto okně je umístěna instance třídy `JTabbedPane`, která obsahuje dvě záložky, v jedné je zobrazován zdrojový kód a v druhém grafický návrh systému. Zatímco pro editaci zdrojového kódu je využívána knihovní třída `JTextPane`, tak o zobrazování i řízení grafického návrhu se stará vlastní třída `DiagramPanel`[1].

Všechna okna, kromě toho hlavního, jsou potomky knihovní třídy `JDialog` a jsou spouštěna jako modální dialogová okna. Nyní popíši okno s názvem *Vytvořit komponentu*, které řídí hlavní třída `NewComponentEditor`. Okno umožňuje uživateli vytvořit vzhled komponenty, deklarovat její porty nebo obnovovat z archivu komponenty již dříve vytvořené. Stromová struktura, ve které je archiv uložen, je zobrazena pomocí komponenty třídy `JTree`. Tento strom modeluje strukturu adresáře `lib\components\` a je vytvářen vždy znovu při každém otevření okna. Při stisku tlačítka *Vybrat z archivu* se z příslušného textového souboru načte zakódovaný popis[2] dané komponenty. Následuje volání metody `setLook()`, která řídí nastavení editačních políček v druhé záložce *Porty* podle načtených parametrů a dále nastavení vykreslované komponenty na panelu třídy `DesignComponentPanel`.

K výběru a zapojování šablon je k dispozici dialogové okno *Vložit šablonu*, které má jako hlavní třídu `TemplateEditor`. V tomto okně se nachází textové pole třídy `JTextPane`, kde je zobrazována šablona ve formátu bez všech kódovacích znaků. Za běhu programu jsou udržovány oba formáty šablony – interní a uživatelská, kde interní formát udržuje starý i

[1] Popisu této třídy je věnovaná kapitola č. 6.1.1

[2] Forma tohoto popisu byla vysvětlena v kapitole č. 5.3

nový název každého ze signálů a při každé změně na pozici nového jména je volána metoda `makeUserTemplate()`, která převádí šablonu na formát srozumitelný uživateli. V okně se nachází tři seznamy třídy `JListBox`, které slouží k výpisu všech signálů entity a signálů objevujících se v šabloně. Pro průchod adresářem je i zde umístěn strom třídy `JTree`, který modeluje stejný adresář `lib\components\` jako archiv komponent v okně *Vytvořit komponentu*. Třída `TemplateEditor` ale obsahuje metodu `controlIfIsTemplate()`, která hlídá, aby byly ve stromě zobrazovány pouze ty soubory, které obsahují i definici chování šablony za klíčovým slovem `$$templatebegin`. Řízení tohoto okna tedy probíhá tak, že uživatel vybere ze seznamů příslušné signály, které hodlá vyměnit, načež metoda `replaceSignal()` vyhledá správné místo, kam nový název vložit. Pokud tedy například vybere signál se starým názvem `input` a novým `data`, proběhne následující operace:

`$s{input}{} -> $s{input}{data}`

Starý název je stále udržován z toho důvodu, aby bylo možné provést např. opravu předchozího prohození názvů, a tak je nutné neztratit původní jméno signálu.

6.1.1 Grafická část editoru

Třída `DiagramPanel` rozšiřuje knihovní třídu `JPanel` a implementuje dvě rozhraní. Tím prvním je rozhraní `Serializable`, díky kterému je možné ukládat libovolné objekty do souboru a poté je znovu rekonstruovat. Tím druhým je rozhraní `MouseMotionListener`, které umí zpracovávat události vyvolávane pohybem myši po panelu. Pro zpracovávání údálostí jako jsou stisky myši je tu rozhraní `MouseListener`. Nejzajímavější a zároveň nejobsáhlejší metody zpracovávající události od myši jsou `mousePressed()`, `mouseClicked()` a `mouseDragged()`. První dvě obsahují poměrně rozsáhlé řídící konstrukce, které analyzují pozici stisku myši a na základě aktuálního stavu volají konkrétní výkonné metody. Metoda `mouseDragged()` zase přepočítává pozice komponent a signálů při přemísťování prvků uživatelem.

Každá komponenta je instancí třídy `ComponentView()`, kde jsou uchovávány kompletní informace o vzhledu komponenty jako například rozmístění portů, pozice na obrazovce atd. a všechny tyto komponenty jsou uloženy do seznamu typu `ArrayList<>`. Základní vzhled komponenty, tedy obdelník určující hranice, tvoří vždy dvě instance knihovní třídy `Rectangle`, jedna zobrazuje hranici a druhá pozadí komponenty. Informace o názvu a typu komponenty a hlavně o všech podrobnostech kolem jednotlivých portů komponenty nese třída `Component`.

Další významnou třídou je třída `Signal`, kde jsou uloženy kompletní informace o každém signálu. Celý signál je složen z libovolného počtu instancí knihovní třídy `Line2D`, tedy úseček, které jsou mezi sebou propojovány a tvoří tak spojitou křivku. Jednotlivé usečky jsou propojovány pomocí zalamovacích bodů, což jsou objekty odvozené od třídy `WrapPoint`, která je vnořenou třídou třídy `Signal`. V každém takovém zalamovacím bodě jsou uchovány indexy všech úseček, které do něj vedou. Úseček může vést do zalamovacího bodu i více než dvě, kdy potom vzniká bod s interním označením `divider`, který reprezentuje bod, kde se signál větví do více směrů. Tento bod je možné libovolně vytvořit na kterékoliv usečce signálu, nicméně editor zde neumožňuje větvení začínat. Dovoluje sem pouze novou větev signálu přivést z portu některé z komponent.

Celkové řízení tvorby a manipulace s diagramem probíhá tak, že při každém stisku myši příslušná metoda rozpozná, na jaký prvek bylo v diagramu zmáčknuto a podle toho nastaví příslušnou globální proměnnou, která udržuje informaci o indexu tohoto prvku

v seznamu objektů stejného typu. Například kombinací indexů uložených v proměnných `currentSignal` a `currentLine` vzniká přesný údaj o umístění konkrétní úsečky atd. Stavy systému mění většinou uživatel pomocí ovládacích tlačítek v panelu nástrojů a na základě těchto stavů se volají příslušné metody a konají se příslušné operace. Za všechny uvedu alespoň `addComponent()`, `addSignal()` nebo `joinSignal()`, která připojuje dvě větve křivek do sebe.

Vykreslování schématu má na starosti metoda `paintComponent()`, ta není volána přímo, ale pomocí metody `repaint()`. Tato metoda je volána při každém stisku myši, jejím pohybu či jakékoliv jiné změně v diagramu.

6.2 Analyzátor a generátor zdrojového kódu

V této kapitole popíši, jakým způsobem v aplikaci funguje generování a analýza zdrojového textu.

6.2.1 Analyzátor

Jak už bylo řečeno výše, tento program neobsahuje žádný lexikální ani syntaktický analyzátor, takže umí v textu pouze vyhledávat určité úseky kódu, ve kterých poté hledá potřebné informace. Program vyžaduje absolutní dodržení syntaktických pravidel jazyka VHDL, pokud je v kódu nějaká syntaktická chyba, analyzátor vypíše na stavový panel chybovou hlášku o neúspěchu analýzy kódu. Analýza kódu probíhá při třech konkrétních situacích:

- Před otevřením okna *Vložit šablonu*, kde analyzátor vyhledává v kódu všechny signály a porty, které se poté zobrazují uživateli v příslušných seznamech třídy `JListBox`.
- Po vytvoření nového souboru a vytvoření základní struktury VHDL, kde program vyhledavá porty, které následně zobrazuje jako grafickou komponentu v grafickém prostředí.
- Před generováním kódu ho program nejprve analyzuje a poté nastavuje indexy, kam se má nově generovaný kód vložit.

Většina metod, které tuto problematiku řeší, využívají v zásadě dva hlavní mechanismy hledání, které jsou voleny různě pro konrétní situace. První spočívá ve využívání metody `indexOf()` knihovní třídy `String`, která vrací index požadovaného řetezce v jiném řetězci. Druhý způsob je analýza kódu pomocí konečného automatu. Zde následuje výčet metod, které se analýzou kódu zabývají:

- `String getDeclarationArchitecture(String sourceCode)` – V zadaném zdrojovém kódu vyhledává deklarační část architektury a vrací ji zpět. Zároveň nastavuje indexy, podle kterých generátor pozná, kam vložit kód.
- `void getSignals(String declarationArchitecture)` – V deklarační části architektury vyhledává deklarace signálů a ukládá je seznamu signálů.
- `String getPortSection(String sourceCode)` – Hledá a vrací zpět sekci **port** entity.
- `void getPorts(String portSection)` – Hledá deklarace portů a ukládá je do seznamu.

6.2.2 Generátor

Generátor využívá metod pro analýzu kódu, které mu hledají pozice pro vkládaný kód. Situace, kdy se do zdrojového textu generuje nový kód, jsou vyjmenovány v následujícím seznamu:

- Generování deklarace konkrétní komponenty jako reakci na uživatelův požadavek, který iniciuje v grafickém prostředí editoru.
- Generování použití konkrétní komponenty, které rovněž iniciuje uživatel v grafickém prostředí.
- Po vytvoření nového souboru se generuje základní struktura VHDL návrhu.

Bylo dbáno na to, aby generovaný kód byl nejen správný, ale i přehledný, takže v každé metodě, která vytváří určitou část kódu, je i část, která analyzuje délky všech řetězců a zarovnává text přehledně pod sebe. Přehled nejzajímavějších metod generující kód následuje nyní:

- `void insertComponent()` – Tato metoda spolupracuje s třídou `DiagramPanel`, když vkládá deklaraci komponenty do zdrojového kódu na základě grafického návrhu vytvořeného v editoru. Rozpozná, pro kterou komponentu v návrhu je operace požadována a vygeneruje do zdrojového textu daný úsek kódu.
- `void insertComponentUsage()` – Opět spolupracuje s třídou `DiagramPanel`, když základě schématu zjišťuje, jak jsou komponenty propojeny, a generuje do zdrojového textu použití komponenty se správně namapovanými porty komponenty.
- `String generateNewFile()` – Metoda je volána po vytvoření nového souboru a generuje základní VHDL strukturu.

Kapitola 7

Testování

Aplikace byla testována v několika směrech, bylo testováno uživatelské rozhraní, generovaný kód a poté i schopnost analyzovat zdrojový kód. Všechny testy probíhaly spíše manuálně a nebyly vyvinuty žádné rozsáhle sady automatických testů. Testovalo se v operačních systémech *Windows 7* a *Ubuntu 11.04*.

Uživatelské rozhraní

Jde o editor, tedy o program, který by měl být k uživateli přívětivý, být přehledný a měl by na pohled pěkně vypadat. K testování těchto vlastností jsem si přizval 4 kolegy, které jsem nechal s programem pracovat bez toho, abych jim předtím vysvětloval, jak funguje. Výsledky takovýchto testů jsou samozřejmě dosti subjektivní, protože každému se může líbit jiný přístup, nicméně každému jsem položil několik otázek, na základě kterých jsem uživatelské rozhraní v rámci možností upravoval. Nejlepší ohlasy byly na vzhled a jednoduchost aplikace, což byl také záměr. Výhrady naopak byly například k některým funkcím grafického editoru jako například nedokonalé mazání prvků, dále pak nemožnost vytvořit svoji vlastní šablonu přímo v editoru.

Generátor kódu

Testování správnosti generovaného kódu probíhalo v prostředí Xilinx ISE, kde bylo ověřováno hlavně, zda daný soubor bude syntetizovatelný. Bylo otestováno mnoho konstrukcí generované programem a bylo objeveno několik chyb, které byly následně opraveny. Nyní program při správném použití dává výsledky, které je syntetizátor schopný zpracovat.

Analyzátor

Třetí věc, která byla testována, je analyzátor kódu. V práci už bylo několikrát napsáno, že program vyžaduje absolutní správnost syntaxe, jinak vypisuje chybovou hlášku, takže testy na chyby v syntaxi v tomto případě nemají smysl. Testovány byly hlavně různé zápisy deklarací signálů a portů, což jsou nejčastější problémy, které analyzátor řeší.

Kapitola 8

Závěr

V rámci bakalářské práce byly nejprve nastudovány a popsány principy jazyka VHDL, jeho syntaxe a sémantika. V práci se také zabývám charakteristikou nejčastěji používaných komponent a způsobem jejich popisu v jazyce VHDL. Důraz byl kladen hlavně na popis pomocí běžně používaných konstrukcí, které jsou vhodné pro syntézu, tedy syntetizovatelných šablon. Poté byl navrhnut a realizován nástroj, jehož použití usnadňuje uživateli mechanickou práci při vytváření, vkládání a propojování jednotlivých komponent návrhu.

Aplikace byla realizována jako tzv. klikátko a důraz byl kladen hlavně na to, aby byl program dostatečně intuitivní a snadno ovladatelný. Právě za tímto účelem byla vytvořena ke klasické textové části i část grafická pro tvorbu schématu. Toto schéma může být využito pro pouhou vizualizaci systému, ale je s ním spojen i mechanismus jeho analýzy a následného generování úseků zdrojového kódu v jazyce VHDL, které toto schéma popisují. Zejména se jedná o analýzu komponent a jejich propojení signály.

Jedním z přínosů této práce je tak vytvoření aplikace, která umí za návrháře odpracovat určité části mechanické práce, které mu zabírají čas při řešení obtížnějších problémů. Program umožňuje uložit si libovolně vytvořenou komponentu, čímž dává uživateli i možnost znovupoužitelnosti svých řešení. Vidím možnost i v použití programu pro začínající návrháře, kteří mohou hledat nástroj, který jim ukáže, jak na to, jak má správně propojování komponent vypadat nebo jak mají vypadat popisy nejčastěji používaných komponent. Mohou tak využívat aplikaci i pro výukové účely.

Možnosti navazujících prací na projektu vidím několik. Vhodným doplněním by byla zcela jistě možnost vytvoření vlastní šablony uživatelem přímo v programu vzhledem k tomu, že nyní je možné využívat pouze šablony předpřipravené a pokud si chce uživatel vytvořit vlastní, musí si ji vytvořit někde mimo program v takovém formátu, aby odpovídal specifikaci.

Prostor je i ve schopnostech analýzy zdrojového kódu. Další vývoj by se mohl zabývat implementací lexikálního a syntaktického analyzátoru a s tím by mohlo být spojené i automatické vykreslování schématu podle libovolného souboru .vhd.

Pro popis hardwaru ve VHDL se často používají řídící konstrukce typu if-then-else, case atd., a tak by mohla další verze programu nabízet možnosti rychlého generování těchto konstrukcí. Dále by mohla obsahovat systém napovídání klíčových slov nebo identifikátorů.

Literatura

[1] Airiau, R.; Bergé, J.; Olive, V.: *Circuit Synthesis with VHDL*. Kluwer Academic Publisher, 1994, iSBN 0-7923-9429-1.

[2] Bhasker, J.: *A VHDL synthesis primer*. Star Galaxy Publishing, druhé vydání, 1998, iSBN 0-9650391-9-6.

[3] Herout, P.: *Učebnice jazyka Java*. Kopp, první vydání, 2001, iSBN 80-7232-115-3.

[4] J. Pinker, M.: *Číslicové systémy a jazyk VHDL*. BEN – technická literatura, první vydání, 2006, iSBN 80-7300-198-5.

[5] Král, J.: *Řešené příklady ve VHDL*. BEN – technická literatura, první vydání, 2010, iSBN 978-80-7300-257-2.

[6] Sekanina, L.: *Návrh počítačových systémů : Studijní opora*. Vysoké učení technické v Brně, 2006.

[7] Vašíček, Z.: Popis HW pomocí VHDL. online, 2009.
URL http://merlin.fit.vutbr.cz/FITkit/docs/navody/synth_templates.html

Příloha A

Obsah CD

/src	zdrojové soubory aplikace
/text	text práce ve formátu .pdf
README.txt	manuál ke spuštění